(11) Publication number: **0 180 077 A3**

Office européen des brevets

EUROPEAN PATENT APPLICATION

(21) Application number: 85112899.1

(22) Date of filing: 11.10.85

(51) Int. Cl.³: G 06 F 9/44

(30) Priority: 31.10.84 US 666800

(43) Date of publication of application: 07.05.86 Bulletin 86/19

(88) Date of deferred publication of search report: 22.06.88

(84) Designated Contracting States: DE FR GB

(71) Applicant: International Business Machines Corporation
Old Orchard Road
Armonk, N.Y. 10504(US)

(72) Inventor: Hopkins, Martin Edward
300 Douglas Road
Chappaqua New York 10514(US)

(72) Inventor: Warren, Henry Stanley, Jr.
Kitchawan Road
Ossining New York 10562(US)

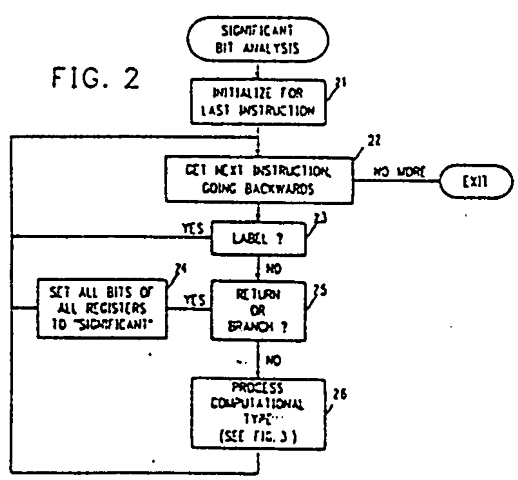
(74) Representative: Appleton, John Edward
IBM United Kingdom Limited Intellectual Property
Department Hursley Park
Winchester Hampshire SO21 2JN(GB)

(54) **A data processing machine for compiling computer programs.**

(57) A data processing machine for compiling a computer program in which a particular operation may be controlled by instructions of different lengths characterised in that the machine is controlled to perform the following operation for each input computer program to be compiled;

1. examining a sequential instruction stream and determining which bits of the result of each instruction could be significant, based on the context of the instruction in the instruction stream, and
2. generating the most efficient instruction form that computes the correct result in only the bit positions that were determined could be significant.

FIG. 2



(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: 0 180 077 A2

EUROPEAN PATENT APPLICATION

(12)

(21) Application number: 85112899.1

(22) Date of filing: 11.10.85

(51) Int. Cl.⁴: G 06 F 9/44

(30) Priority: 31.10.84 US 666800

(43) Date of publication of application: 07.05.86 Bulletin 86/19

(84) Designated Contracting States: DE FR GB

(71) Applicant: International Business Machines Corporation, Old Orchard Road, Armonk, N.Y. 10504 (US)

(72) Inventor: Hopkins, Martin Edward, 300 Douglas Road, Chappaqua New York 10514 (US)
Inventor: Warren, Henry Stanley, Jr., Kitchawan Road, Ossining New York 10562 (US)

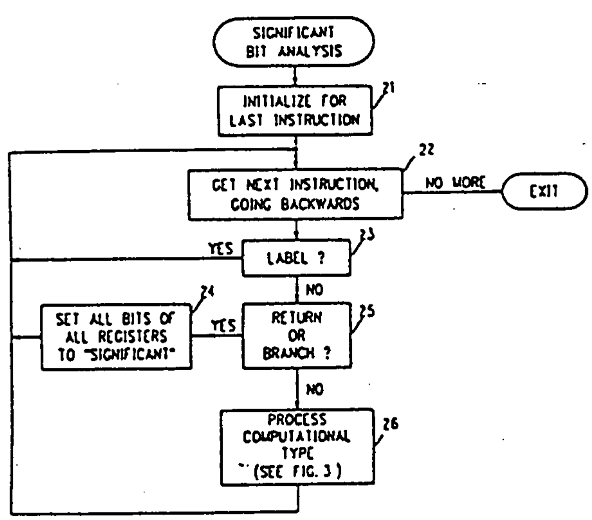
(74) Representative: Appleton, John Edward, IBM United Kingdom Patent Operations Hursley Park, Winchester, Hants, SO21 2JN (GB)

(54) **A data processing machine for compiling computer programs.**

(57) A data processing machine for compiling a computer program in which a particular operation may be controlled by instructions of different lengths characterised in that the machine is controlled to perform the following operation for each input computer program to be compiled;

1. examining a sequential instruction stream and determining which bits of the result of each instruction could be significant, based on the context of the instruction in the instruction stream, and

2. generating the most efficient instruction form that computes the correct result in only the bit positions that were determined could be significant.



EP 0 180 077 A2

ACTORUM AG

A DATA PROCESSING MACHINE FOR COMPILING COMPUTER PROGRAMS

This invention relates to data processing machines that are used to compile computer programs and produce object code from a source code input.

This invention has particular utility in a compiler for a digital computer in which optimization algorithms are used to improve the quality of the code. It could also be used in an assembler, although optimizing assemblers are not common. This invention improves the quality of the object code generated by the compiler or assembler.

The invention is applicable to computers that use an accumulator or a set of general registers, and that have one or more instruction types that exist in two or more forms, with the forms differing in the length of the operands. Furthermore, for this invention to be applicable there must be some reason to prefer a shorter form on an instruction, when it can be used. On many machines, the shorter forms are preferable because they execute faster, or they occupy less storage, or both.

The quality of code produced by compilers has been an issue ever since the first compiler was produced. One of the principal objectives of IBM's FORTRAN I compiler, the first commercially available compiler, was to produce object code in the field of scientific computations which was comparable in code quality to that produced directly by assembly language programmers coding "by hand".

Today, higher level languages are designed to be used in every field in which computers are applicable. Even the original FORTRAN language has been bolstered to make it applicable to a wide range of programming tasks. However, it is still important that the quality of code produced by the compiler be high, especially if the resultant code is to be used in a production environment. Code produced by a skilled assembly

language programmer is still the yardstick against which compiler produced code is measured.

A large number of optimization technique have been developed and refined since the 1950's to improve the quality of compiler generated code. Indeed, many of these optimizations where known in principle, and used in some fashion by the team that produced the first FORTRAN compiler.

Optimizations that are frequently employed in optimizing compilers can be divided into two classes, which are commonly known as "global" and "peephole" optimizations. Global optimizations are those that are based on an analysis of the entire program being complied. Examples are "code motion" (moving code out of loops) and "common subexpressions elimination." Peephole optimizations are those that are based on an analysis of a relatively small region of the program, such as a "basic block", or perhaps only two adjacent instructions.

The present invention can be implemented as a global optimization, or at the basic block level, or it can use partial global information that is normally readily available from global analysis, and then refine this information at the basic block level. With more information available, it is more effective. The information from global analysis that it can profitably use is that of live/dead information. This information tells, for each register operand of an instruction, whether or not that register can be used again before it is reloaded with a new quantity.

If a register operand of an instruction is "dead", that means that the instruction is the "last use" of the register, and after executing the instruction the contents of the register could be altered in any way without affecting the execution of the program. If a register operand of an instruction is "live", on the other hand, that means that the contents of the register cannot be altered after execution of the instruction, because there may be an execution path in which the register's contents are used again.

The following references discuss live variable analysis:

J D Ullman, A Survey of Data Flow Analysis Techniques, Second USA-Japan Computer Conference Proceedings, AFIPS Press, Montvale, New Jersey, (1975), pp 335-342 (contains 37 references).

A V Aho and J D Ullman, Principles of Compiler Design, Addison-Wesley, (1977).

M S Hecht, Flow Analysis of Computer Programs, Elsevier North-Holland, New York, (1977).

The Motorola MC68000 is an example of a type of computer to which this invention is applicable. This computer has three forms of "add", and three forms of "add immediate," as illustrated below.

```
ADD.L r1,r2        ADDI.L #123,r1
ADD.W r1,r2        ADDI.W #123,r1
ADD.B r1,r2        ADDI.B #123,r1
```

ADD.L (add long) adds the entire 32-bit contents of register r1 to register r2, and places the result in r2. ADD.W (add word) adds the rightmost 16 bits of r1 to the rightmost 16 bits of r2, leaving the leftmost 16 bits of r2 unchanged. ADD.B (add byte) adds the rightmost eight bits of r1 to the rightmost eight bits of r2, leaving the leftmost 24 bits of r2 unchanged. Similarly, ADDI/L (add immediate long) adds a number (123 is shown) to the entire 32-bit contents of register r1, ADDI.W adds to the rightmost 16 bits, and ADDI.B adds to the rightmost eight bits.

The instructions ADD.W and ADD.B execute faster than ADD.L, and hence are preferred in a situation in which either would do. The instructions ADDI/W and ADDI.B execute faster and occupy less storage than ADDI/L, and hence are preferred to ADDI/L.

The Motorola MC68000 has many other instruction types that exist in "long" and "short" forms, with the shorter form being faster in execution and often occupying less storage. Further details, including instruction timings, may be found in:--

MC68000 16-bit Microprocessor User's Manual, Second edition, Motorola, Inc., (January 1980).

According to the invention there is provided a data processing machine for compiling a computer program in which a particular operation may be controlled by instructions of different lengths characterised in that the machine is controlled to perform the following operation for each input computer program to be compiled;

1. defining a self contained sequence of instructions,

2. reversing the order of the sequence,

3. for each instructions that may have a different length, in turn determining the maximum register bit positions required to store the significant digits of the output that would result from executing this instruction.

4. marking the instruction to indicate this maximum length

5. replacing in the sequence any instruction found to require less register lengths than originally defined with an instruction of the appropriate definition.

As an example of the code improvement accomplished by this invention, suppose a compiler has generated the instruction sequence:

```
ADD.L     R2,R1
SUBI/L    #16,r1
MOVE.W    r1,6(r5)
```

and suppose further that the MOVE.W instruction, which stores the rightmost 16 bits of register r1 into storage at a location addressed by the contents of register r5 plus 6, is the last use of register r1. Then this invention will replace the ADD.L instruction with ADD.W, and the SUBI.L instruction with SUBI.W. The latter forms execute faster than the former, and the SUBI.W instruction occupies less storage than SUBI.L.

In order that the invention may be fully understood a preferred embodiment thereof will now be described with reference to the accompanying drawings in which:

FIG. 1 is a very high level functional flowchart of an optimizing compiler in which the present invention has particular utility.

FIG. 2 is a high level flowchart of the herein disclosed compiler module for effecting the desired significant bit analysis.

FIG. 3 is a more detailed flowchart illustrating how some of the computational type of instructions are processed by the herein disclosed compiler module.

The invention will be described as if fits into an optimizing compiler, and for the Motorola MC68000 target machine.

The first step in applying this invention is to do "significant bit analysis" of the program being compiled. This is a process of determining, for each instruction, which bits of the result of that instruction are "significant" (it is assumed, for simplicity, that there is only one result). A bit is significant if it is used by the program after being computed. This is the same concept as the well known "liveness," except that each bit of the result is examined to determined whether or not it is "live," or "significant". In conventional liveness analysis, only a single summary bit is computed for each instruction,

which indicates whether or not any bits of the result are "live", or "significant".

There are three levels at which significance analysis could be done:

1. Globally, as ordinary liveness analysis is done,

2. On a basic block (actually branch-to-branch) level, with assistance from global liveness analysis, or

3. On a basic block (or branch-to-branch) level with no assistance.

The first choice above is the most expensive to compute but gives the best results. The last is the cheapest to compute but gives the poorest results. The second is a compromise that is only a little more expensive to compute than (3) if liveness analysis has been done anyway, and gives results of intermediate quality. Choice (2) will be described herein, however the invention is not intended to be limited to this particular approach.

FIG. 1 shows where level (2) of significant bit analysis (block 5) is most conveniently done within the framework of a typical optimizing compiler. The important thing is that global code optimization (block 3) is done before significant bit analysis. This is because conventional liveness analysis is part of global code optimization, and we will need the "last use" bits that are a byproduct of liveness analysis.

It is preferable, although not necessary, that register allocation (block 4) be done before significant bit analysis. This permits a more efficient compiler, because if register allocation is done first, significant bit analysis can be done in terms of the real registers of the machine, and there is usually a fairly small number of them (e.g., 16 on the Motorola MC68000).

Significant bit analysis has to be done before final code generation (block 6). The final code generation module of the compiler will use the results of significant bit analysis to determine what form (8-, 16-, or 32-bit) of each instruction to generate.

FIG. 2 is a high level flowchart of the significant bit analysis shown in block 5 of FIG. 1. The analysis is done in a single backwards pass over the program. Although the processing shown here could be done on a basic block basis, it is shown being done on a branch-to-branch basis. This is just as easy to program, and it will sometimes result in better quality code being generated. Thus label points in the program are ignored (block 23). However, branch instructions (block 25) result in resetting the machine's knowledge of which bits are significant in each register, to the state "all bits of all registers are presumed to be significant." This is the safe state to assume of those pints at which the machine has no information.

If an instruction is not one that represents a label point, and is not a branch or subroutine "return" instruction, then it is an ordinary computational instruction such as "add," "load," "store," "shift," etc. The processing of computational instructions, shown as block 26 of FIG. 2, is shown in more detail in FIG. 3.

This program works with an array of bits referred to herein as the significant bit table (SBT), that has a number of rows equal to the number of registers on the machine (e.g., 16 on the Motorola MC68000), and a number of columns equal to the register length of the machine (32 on the MC68000). At a typical point in processing, the array might look like this:

32 bits

0.	0000FFFF	8.	FFFFFFFF
1.	FFFFFFFF	9.	0000FFFF
2.	FFFFFFFF	10.	000001FF
3.	FFFFFFFF	11.	7FFFFFFF
4.	FFFFFFF0	12.	0F0F0F0F
5.	000000FF	13.	0000FF00
6.	FFFFFFFF	14.	FFFFFFFF
7.	FFFFFFFF	15.	FFFFFFFF

Here we have shown the bits in hexadecimal notation, e.g., "0000FFFF", denotes 16 zero-bits followed by 16 one-bits. The values in the array change as the program scans backwards in the instruction stream. If, at a certain point, the array has the values shown above, then the meaning is that at that point, the leftmost 16 bits of register 0 are not significant ("dead"), but the rightmost 16 bits are significant. A value of "FFFFFFFF" means that all bits in the associated register are significant, etc.

Now, with reference to FIG. 3, let us see how an instruction is processed to determine the significant bits of its result. The process is to propagate the significant bits from a result to the input operands of an instruction. Then, it is propagated from the input operands of a current instruction to the result operands of earlier instructions, as the program controls the processor to scan backwards through the instruction stream. Just how the bits propagate from a result to the input operands depends on the instruction type (add, shift, store, etc.), as shown in FIG. 3. To get started, the machine must know, or assume, which bits of the result of the first encountered instruction are significant. For the process being described, the first encountered instruction is assumed to have all its result bits significant. This is recorded by initializing the entire 16x32 bit array to all one's when a branch instruction is encountered.

Now, suppose the middle of a branch-to-branch section of code is being currently processed, and an add or substrate or subtract instruction is encountered. In particular, suppose the instruction is:

```
ADD.L r1,r2
```

This means to add the contents of register r1 to the contents of register r2. Register r2 is both an input to and the result of the instruction. It will be easier for us to think in terms of a three-address instruction:

```
ADD.L R1,r2,3
```

in which r3 is the result register

First it is necessary to refer to the significant bit table at position r3, to see what bits of the result are significant. The bit mask retrieved from the table is associated (stored) with the add instruction, so that it can be used later by the assembly and final code generation module to generate the optimum form of the add instruction. Actually, for efficiency it suffices to associate only two bits with the add instruction, to record whether the instruction should be generated in long (32-bit) form, word (16-bit) form, or byte (8-bit) form, as those are the only choices on the MC68000. This association of two bits with the instruction will be referred to subsequently as "marking" the instruction.

Suppose the significant bits of the result register r3 (as determined for the SBT) are X700008012'. Then we can mark the add instruction as "word," or 16-bit, form, because all significant bits of the result lie in the rightmost 16 bits of register r3. Then, since addition is a right-to-left process (two's complement arithmetic is assumed throughout), the bits in the leftmost 16 positions of registers r1 and r2 cannot possibly affect a significant bit of the result, but bits anywhere in the rightmost 16 positions can. Therefore, the significant

bits of registers r1 and r2 for this instruction are X'0000FFFF'. This is next recorded in the SBT table, at the rows for registers r1 and r2. If the add instruction is the last use of r1 (or r2) in that block, then the table position for r1 (or r2) is set to X'0000FFFF'. This is determined by looking at the liveness bit described above which may be set during the "Code Optimization" phase. On the other hand, if the add instruction is not the last use of r1 (or r2), then we "OR" X'0000FFFF' into the table at position r1 (or r2). The point is that r1 and r2 may have certain bits significant because of uses of these registers below the add instruction, e.e., uses that were processed earlier in this backwards scan, and wherein those significant uses must not be "forgotten". This processing of an add (or subtract) instruction is shown in blocks 31 and 32 of FIG. 3.

As the backwards scan proceeds, it will likely come to an instruction that sets r1 to r2. At this point, it refers to the table at position r1 or r2, respectively, to determine which bits of that register are significant. It then propagates this information back to the input operands, in a manner similar to the processing of the add instruction described above.

Suppose as another example that a "store byte" instruction is encountered (FIG. 3 blocks 39 and 40). This instruction would be written:

```
MOVE.B r1,d(r2,r3)
```

in which register r1 contains the byte being stored, and r2 and r3 are "base" and "index" registers that serve to address storage. "d" is a displacement (a constant) that has no role in significance analysis. The MOVE instruction has no result register (it doesn't alter the contents of any register). It uses only the rightmost eight bits of register r1. Therefore, a mask of X'000000FF' is OR'ed into the table at the position of r1. The MOVE instruction uses the rightmost 24 bits

of the base and index registers, so a mask of X'00FFFFFF' is OR'ed into the table at the positions of r2 and r3.

FIG. 3 shows the processing of six instruction types. The complete process should be expanded to include other instruction types not shown in FIG. 3, such as "load" and "shift" instructions, etc.

To now present a more substantial example, suppose that a sequence of code between branches is:

		significance of r1
MOVE.L	4(r2),r1	0000FF00
LSR.L	#8,r1	000000FF
ANDI.L	X'000000FF',r1	000000FF
MOVE.B	r1',0(r2)	

The program reads a long word (32 bits) from memory, shifts it right eight positions (LSR = logical shift right), "AND's" it with the mask X'000000FF', and stores the rightmost byte of register r1 into main memory. The column headed "significance of r1" shows one row of the significance array, that for r1, as processing proceeds backwards. The following will describe what happens to the significance of r1 as this sequence of instructions is processed.

Initially (bottom row), the significance of r1 is set to X'FFFFFFFF', which is what has to be assumed in the absence of any knowledge. Then the MOVE.B instruction is encountered. For this example, assume that the use of r1 in this instruction is flagged as a "last use," which has been denoted with a prime (') after r1. Then the significance of r1 is set to X'000000FF' in the table, following FIG. 3 block 40.

Next the ANDI.L is encountered. This instruction uses r1 as both an input and the result register. The significance of r1 as a result, X'000000FF', is "AND'd" with the mask, also X'000000FF' is "OR'ed" into

the table for position r1. The result is X'000000FF' (no change to the significance of r1). These steps are summarized in FIG. 3 block 42.

Now at this point, the significance analysis program could observe that the "NAD" instruction turns off only insignificant bits, and hence can be omitted. Alternatively, the instruction could be marked as byte form, and final code generation could delete it, since the immediate mask ends in eight "1" bits.

Next the LSR.L is encountered. It is marked as byte form, because the significance of the result is X'000000FF'. The significance of the input register, r1, is the significance of the result register, also r1, shifted left eight positions (handling of shifts is not shown in FIG. 3).

Lastly, the MOVE.L is encountered. This is marked as word (16-bit) form, because the significance of the result register (r1) is X'0000FF00', i.e., only bits in the rightmost 16 positions are significant.

By using the marking computed above, final code generation can output the following instructions as a faster and shorter equivalent to those shown above:

```
MOVE.W    6(r2),r1      (load instruction from memory)
LSR.W     #8,r1
MOVE.B    r1',0(r2)     (store instruction to memory)
```

There are two things that final code generation must be cognizant of that arise in the above example: (1) the MOVE.L cannot be changed to MOVE.W unless the increase of two in the displacement (from 4 to 6 in the example) results in a displacement within the limits allowed by the instruction (32767 for the MC68000), and (2) the selection for the most efficient form of the LSR instruction depends upon the significance of the result and the shift amount. In the example, the significance of

the result is X'000000FF', but the LSR instruction cannot be made LSR.B, because bits in positions 16-23 of register r1 are shifted into positions 24.31. It can, however, be made LSR.W, which is faster than the original LSR.L.

Appendix I shows a complete working subroutine for performing significance analysis. It includes the steps that were illustrated in FIG. 2 and FIG. 3. It has been utilized successfully with several target machines: the Motorola MC68000, the IBM System/370, and several experimental reduced instruction set machine architectures. It is written in a language similar to PL/I, and is sufficiently annotated to allow any skilled programmer to incorporate the present invention into an optimizing compiler of the form shown in FIG. 2, or to rewrite the subroutine in another language, or for another target machine, or for a compiler to assembly of different structure.

It will of course be appreciated that the specific significance values introduced into the bit table would have to be tailored to a specific system architecture and particular instruction format. The invention would have equal applicability to, for example, a 16 bit full word machine as well as to a 45 to 64 bit machine architecture. It should be noted more particularly, that the advantages of the present invention will be realized in a machine architecture where the shorter form instructions take less machine time or less storage space than the longer form regardless of the underlying and length of the machine.

CLAIMS

1. A data processing machine for compiling a computer program in which a particular operation may be controlled by instructions of different lengths characterised in that the machine is controlled to perform the following operation for each input computer program to be compiled;

1. defining a self contained sequence of instructions,

2. reversing the order of the sequence,

3. for each instructions that may have a different length, in turn determining the maximum register bit positions required to store the significant digits of the output that would result from executing this instruction.

4. marking the instruction to indicate this maximum length

5. replacing in the sequence any instruction found to require less register lengths than originally defined with an instruction of the appropriate definition.

2. A data processing machine for compiling a computer program in which a particular operation may be controlled by instructions of different lengths characterised in that the machine is controlled to perform the following operation for each input computer program to be compiled;

1. examining a sequential instruction stream and determining which bits of the result of each instruction could be significant, based on the context of the instruction in the instruction stream, and

2. generating the most efficient instruction form that computes the correct result in only the bit positions that were determined could be significant.

3. A data processing machine as claimed in claim 2 in which the machine is controlled to

1. examine a linear instruction stream in reverse order and determining for each-instruction those bits of the result which must be considered significant (significance value (s.v.)) and marking each input operand to that instruction to the same significance value,

2. propagate upwards in the instruction stream said input operand significance values to any instructions wherein the result of said instruction is an input operand to a subsequent instruction,

3. continue said procedure in reverse order until all of the instructions in the instruction stream have been evaluated and generating the most efficient instruction form for the entire stream for those instructions whose result significance value were changed from an "all bits significant" state by said evaluation procedure.

4. A data processing machine as claimed in claim 3 wherein real registers are assigned to each instruction and controlled to a (s.v.) memory location relating to each real system register for storing significance value data for operands to be stored in each said register,

set each said (s.v.) memory location to a predetermined maximum at the beginning of said procedure,

and change the significance value of a particular memory location when the operand to be stored in the related register has a significance value other than that already present.

5. A data processing machine as claimed in claim 4 further controlled

after completion of an evaluation procedure,

to access said (s.v.) memory at the assigned locations relating to the result of each instruction and marking said instruction with the significance value stored in said table for use in the "final code generation" phase of the compiler.

6. A data processing machine as claimed in claim 5 further controlled to determine if the use of a particular register is a "last use" in the instruction stream being evaluated before entering a significance value in the (s.v.) memory location related to said register,

substitute the new value for the old value if it is a "last use," and

'OR' the new value with the old value if it is not a "last use".

7. A code optimizing method operable within an optimizing compiler after the register allocation phase thereof,

said method being operable to perform significant bit analysis, the results of said analysis being utilized during the final code generation phase of the compiler operations for generating the most efficient form of computer instruction available for producing the required result said method comprising,

establishing a significant bit table in memory being a storage location relating to each working register for target system and means for accessing said table at a location derivable from the register number assignment of the operands of an instruction,

examining a linear code stream which is to be optimized, sequentially in reverse order,

if an instruction being evaluated is a non-computational "label" obtaining the next instruction,

if an instruction is a return or branch instruction setting the significance value of all of the array storage locations to "all significant" and obtaining the next instruction,

determining for each computational type instruction, the maximum significance value of its result and specifying this significance value as the required significance values for its two input operands,

determining the significance value for a particular result as being no greater than the significance value required for a subsequent operation to which it will become an input,

and continuing the evaluation process until all of the instructions in a particular instruction stream have been processed,

and marking each instruction with a significance value marker representative of the results of said significant value process.

8. A method as claimed in claim 7 wherein an 'add' or 'subtract' instruction is obtained and which comprises setting or adding to the significant bits of each input register all the bits at and to the right of the leftmost significant bit of the result register, in said significant bit table.

9. A method as claimed in claim 7 or claim 8 wherein the computational instruction comprises a "move register" or an "AND, OR, or EXOR" logical operation" including;

setting or adding to the significant bits of each input register of the instruction, the significant bits of the result register.

10. A method as claimed in claim 7, 8 or 9 wherein the computational instruction comprises a "compare registers" operation including;

setting the significant bits of each input register as "all significant."

11. A method as claimed in any one of claims 7 to 10 wherein the computational instruction comprises a "store word" operation including;

setting or adding a significance value to the significant bit table at the location relating to the register to be stored equal to the field length of a word in the target system and setting or adding a significance value to the significant bit table at locations related to the base and index registers, a value equal to the field length of the base and index fields required by the target system memory.

12. A method as claimed in any one of claims 7 to 11 wherein the computational instruction comprises a "store byte" operation including;

setting or adding a significance value to the significance bit table at the location relating to the register to be stored equal to the field length of a byte in the target system and setting or adding a significance value to the significant bit table locations related to the base and index registers, a value equal to the field length of the base and index field required by the target system memory.

13. A method as claimed in any one of claims 7 to 12 wherein the computational instruction comprises an "AND immediate" instruction including;

setting or adding to the significance value in the significant bit table at the location relating to the input register, the significance value of the result of the instruction, 'AND'ed' with the "immediate mask."

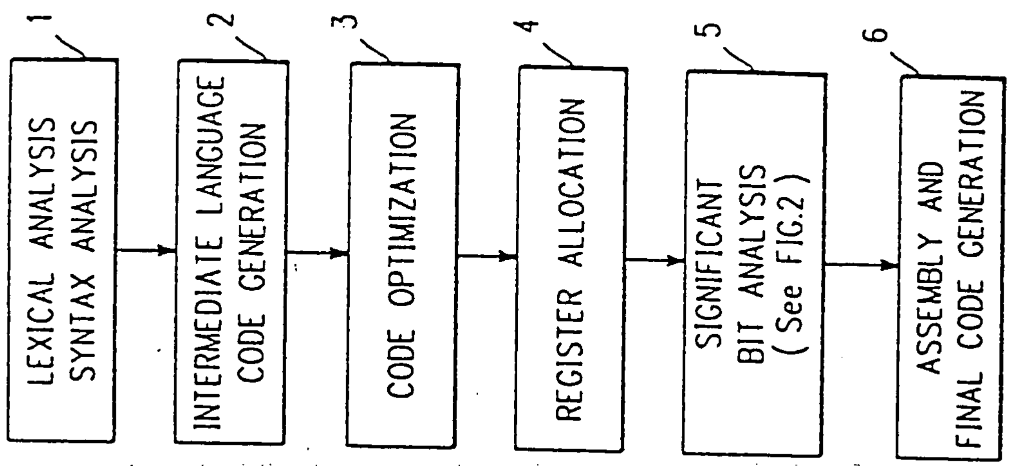
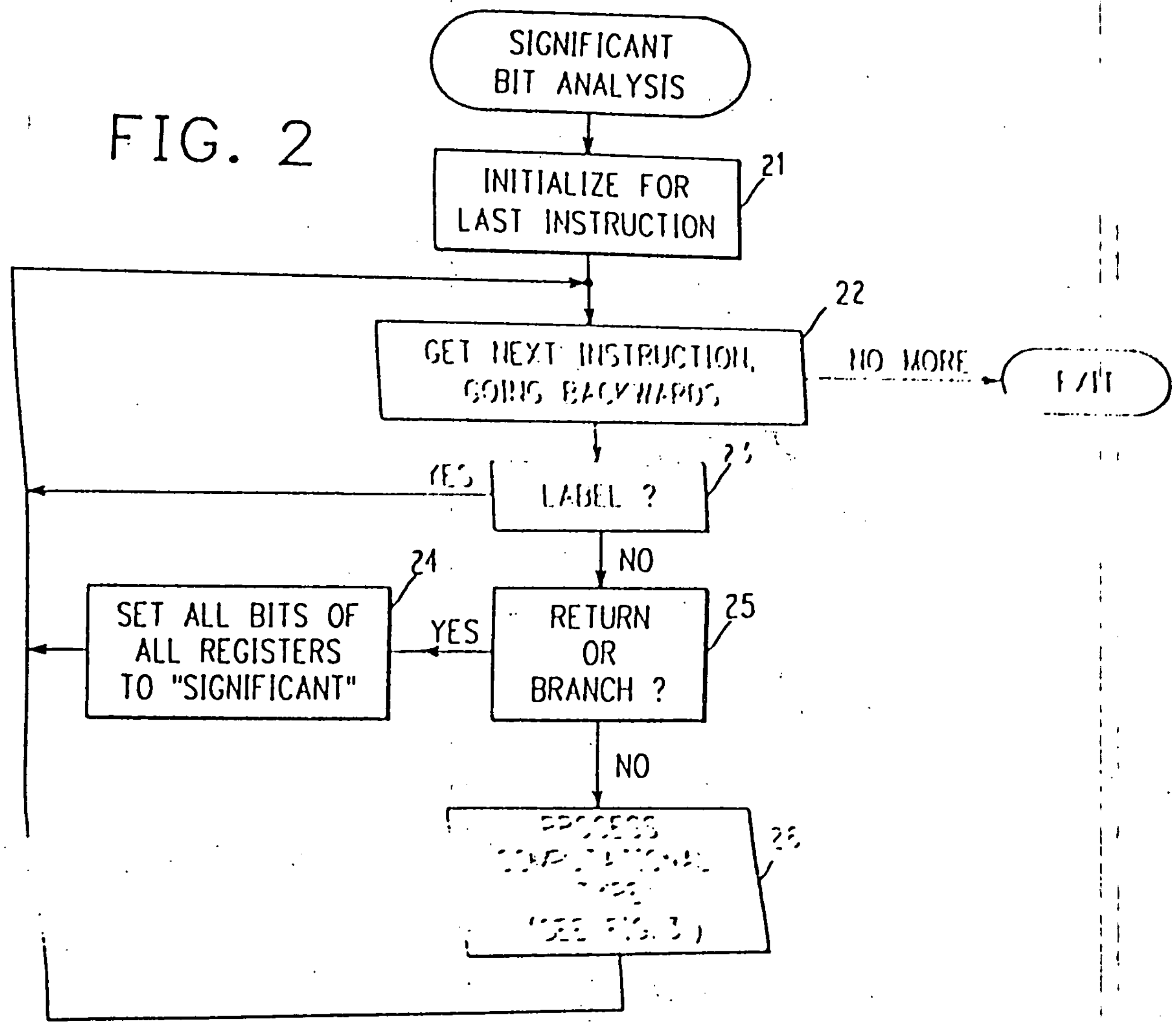

1
LEXICAL ANALYSIS
SYNTAX ANALYSIS
2
INTERMEDIATE LANGUAGE
CODE GENERATION
3
CODE OPTIMIZATION
4
REGISTER ALLOCATION
5
SIGNIFICANT
BIT ANALYSIS
(See FIG.2)
6
ASSEMBLY AND
FINAL CODE GENERATION

FIG. 1

FIG. 2


SIGNIFICANT BIT ANALYSIS
INITIALIZE FOR LAST INSTRUCTION
21
GET NEXT INSTRUCTION, GOING BACKWARDS
22
NO MORE
EXIT
LABEL ?
YES
NO
RETURN OR BRANCH ?
25
YES
SET ALL BITS OF ALL REGISTERS TO "SIGNIFICANT"
24
NO

European Patent Office

EUROPEAN SEARCH REPORT

0180077

Application Number

EP 85 11 2899

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.4)
A	W.M. WAITE & G. GOOS: "Compiler Construction", 1984, pages 342-353, Springer-Verlag, New York, US * Section 13.3.1 *	1,2,7	G 06 F 9/44
A	COMMUNICATIONS OF THE ACM, vol. 19, no. 3, March 1976, pages 137-146, Association for Computing Machinery, Inc.; F.E. ALLEN et al.: "A program data flow analysis procedure" * Page 138, right-hand column, line 55 - page 139, left-hand column, line 16 *	1,2,7	

TECHNICAL FIELDS SEARCHED (Int. Cl.4)

G 06 F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	18-01-1988	WILTINK J.G.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0401)